DEPARTMENT OF DEFENCE

DSTO

Information Technology Division

RESEARCH NOTE
ERL-0663-RN

AN *INTERIM REPORT ON THE DISTRIBUTED DATABASE* DYNAMIC RECONFIGURATION PROJECT

by

Damian O'Dea

ELECTRONICS RESEARCH LABORATORY

APPROVED FOR PUBLIC RELEASE

AR-007-015

DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

RESEARCH NOTE
ERL-0663-RN

AN INTERIM REPORT ON THE DISTRIBUTED DATABASE DYNAMIC RECONFIGURATION PROJECT

by

Damian O'Dea

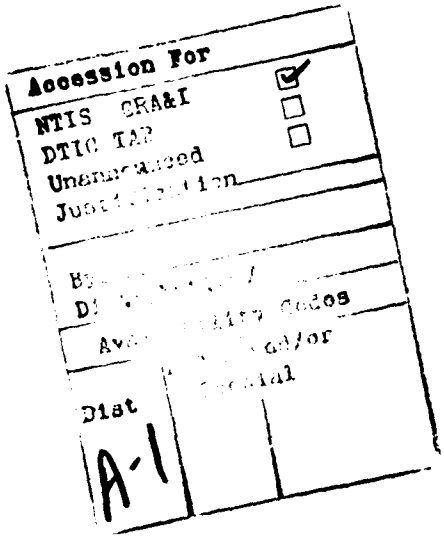
SUMMARY

This paper discusses the work which took place in the first phase of the development of the D^3R project. The concepts of Object-Oriented Programming and Multitasking are introduced and explained. Each stage of the project is detailed, and where relevant, concepts from the DBMES system are introduced and explained. The paper concludes with recommendations for future developments to both systems as they approach completion.

JAN 93

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory,PO Box 1500, Salisbury, South Australia,5108.



CONTENTS

FIGURES

APPENDICES

ABBREVIATIONS

ALDM	ADDAM-Like Distributed Database Manager
AMD	Advanced Micro Devices
C3ISE	Command, Control, Communications and Intelligence Systems Engineering Group
CPU	Central Processing Unit
D^3R	Distributed Database Dynamic Reconfiguration
DBE	Database Element
DBM	Distributed Database Management
DBMES	Distributed Database Management Evaluation Software
DBX	Distributed Database Exerciser
DPTBS	Distributed Processing Testbed System
FDDI	Fiber Distributed Data Interface
FIFO	First-In, First-Out
GDP	Graphics Demonstration Package
IDE	Integrated Development Environment
ITD	Information Technology Division
LIFO	Last-In First-Out
OO	Object Oriented
RTG	Real-Time Graphics and Measurement Control Tools
SR2PC	Single Record Two-Phase Commit
TC++	Turbo C++
VRTX	Versatile Real-Time Executive
VT	Virtual Terminal

1 INTRODUCTION

The Distributed Database Dynamic Reconfiguration (D^3R) project is a part of the Distributed Processing for Combat Systems task (NAV 87/226.3). D^3R is investigating methods of detecting and correcting failures of Distributed Database systems. In real life, failures are caused by faulty networks, faulty processing components and operator errors. These failures can result in incorrect data leading to invalid results. This may cause the system to fail completely, or produce faulty conclusions.

The D^3R project will form a component of the Distributed Processing Testbed System (DPTBS)[1], which is comprised of the Distributed Database Exerciser (DBX), the ADDAM-Like Distributed Database Manager (ALDM)[2] and the Distributed Database Management Evaluation Software (DBMES)[3]. The components of this system run on IBM PCs connected by a fibre optic network. Access to the network is provided by hardware cards (plug-in circuit boards with special components) with fibre optic connections (for more detail cf. section 2.1). Programs can communicate across the network through the use of specially provided library routines. These are pre-written software routines which have been built to interface with the hardware cards. A library is a simple way of grouping these different routines into a single, compact and convenient file.

The DBX is a database exerciser made up of three modules: a Simulator to generate simulated sensor data, a Tracker to resolve this data into target tracks, and Engager to manipulate track data [1].

The ALDM implements a distributed database manager. By providing a partially replicated database, it allows all nodes (machines on the network) which require access to some elements of the database to store a local copy of that data. This greatly improves access time, allowing faster operations. ALDM provides two forms of data update, namely "performance", which is a high speed update used for short-lifespan, repetitive data, and "reliable", which is slower than performance, but guarantees that an update will occur for all copies of a data element. Reliable updates maintain global database consistency, while performance updates rely on any erroneous data being quickly replaced with a correct copy at the next update [2].

The DBMES is an evaluation package for distributed database protocols which is used to obtain data about how well the system is performing under various conditions. The DBMES is easily modifiable for use with several different database management protocols. In addition to exercising the database for evaluation purposes, a Graphics Demonstration Package (GDP) has been created to show the database operating using graphical displays [3, 4].

This report covers work undertaken on the D^3R project in the period spanning January to March 1992. Further work in this project will be undertaken in 1993.

2 OVERVIEW

In the execution lifetime of any program which requires access to a network, there is always a possibility that the system can fail in some way. With primitive systems, programs which require this access may cease to work due to only a single fault on the network, damaging any further access.

Distributed systems are more robust, and a network fault can be tolerated to some degree. In some cases, such as the Fiber Distributed Data Interface (FDDI) dual-ring network, the failure of a station (ie. machine on the system, or "node") means that that station is simply bypassed (cut out of the ring), while network operations continue normally. Failure of two non-adjacent stations, however, leads to network partitioning, because of the way in which a node is bypassed. This self-healing process is called "wrapping". Essentially the stations on either side of the disfunctional node will terminate the ring and feed information back to the preceeding

node on the secondary ring. With two disfunctional nodes, the ring breaks into two "partitions," or two "half-networks," both of which continue to run normally, except that they are unable to pass data to each other until the faults are repaired.

When this situation arises, it is necessary for a distributed system to be able to continue to execute correctly. In the case of the ALDM, the D^{3}R protocol is intended to maintain database consistency, both when partitioning is detected, and when a network is repaired.

The ALDM database is partially replicated. Multiple copies of each page (ie a logical division or partition of the database) are stored on several, but not all, nodes of the system. Thus the overall database is actually a gestalt of the multiple copies of all of its pages.

Once partitioning occurs, there is a high probability that some of the page copies in one partition will be modified, but the copies in the other partition will not, because the node requesting the update will reside in only one partition of the network. The database is then deemed to be "inconsistent," because of the discrepancies between page copies. When the two partitions are rejoined, decisions must be made as to which partition's database page copy is accurate, so that the other partition's copies can be corrected to match that data. The operations carried out by such a protocol when making these decisions are termed "dynamic reconfiguration of a distributed database."

The D^{3}R program is based on software written by C3ISE Group/ITD for the DBMES package. The evaluation package is also being used as a driver for a simple Distributed Database Management (DBM) protocol, which will be used to support and test the reconfiguration protocol. The more complex ALDM protocol will replace the simpler DBM at a later date. As much of the design of the D^{3}R is similar to that of the DBMES, some time will now be spent describing how that package was constructed [4].

2.1 Platform

The platform for all Distributed Processing for Combat Systems projects is a network of IBM PC compatibles (henceforth "PCs" for simplicity) connected by a fibre optic network controlled by Advanced Micro Devices' (AMD) Fiber Distributed Data Interface (FDDI) Fastcards. The programs running on the platform interface to the Fastcard hardware using several library routines provided by AMD. This platform is called the DPTBS.

The DBMES program is an Object Oriented (or "OO") system written in C++. The compiler used is part of Borland International's Turbo C++ (TC++) Integrated Development Environment (IDE). The IDE provides an editor, compiler, linker, and debugger all in one system. The evaluation software package utilises interfaces to TC++ standard libraries, to Ready System's Versatile Real-Time Executive (a multitasking kernel, aka "VRTX") library, and to AMD's FDDI interface library.

The Graphics Demonstration Package (GDP) additions to the DBMES (mentioned above) utilise an interface library to graphics routines provided by Quinn-Curtis' Real-Time Graphics and Measurement Control Tools (RTG) product.

During program development of both the DBMES and D^{3}R, a Virtual Terminal (VT) was connected to one PC. The VT displays data which is sent to it from the PC via a serial port connection. This allows error and report messages to be displayed by the development programs for diagnostics and debugging purposes, without cluttering the normal display of the program output. The different function call used to achieve error and report message output on the VT also makes the removal of this debugging code easier.

2.2 Object Orientation

The DBMES is an Object Oriented (OO) system. This means that the paradigm used in it's design and development is quite different from programs written with the more traditional procedural paradigm. OO takes several concepts which have been available to Computer Science for some time and unites them into a powerful programming methodology.

OO has the benefits of being more natural for modelling complex real-world systems in a program, and of resulting in code which is often re-usable in different programs. One possible drawback of OO is the difficulty of learning the new paradigm, due to the conceptual shift. Once this is mastered, however, my own experience has shown that OO is a far better method of programming distributed systems than the procedural norm.

The central concept of the OO methodology is that of an object. The object device allows abstraction away from the details of complex operations, encapsulation (hiding) of those same details, hierarchical organisation of these abstractions in increasing levels of complexity, and the treatment of similar abstractions by the same methods.

The C++ language provides facilities for the definition and usage of classes. Each object in a program belongs to a certain type, and the definition of this type is called a "class." Thus, in the same way we have constructs for user-defined types (such as record "structures"), we are able to create user-defined classes. The instantiation (ie. actual storage space set aside for a unit of the defining type during a program execution) of a given type is a "variable," and similarly the instantiation of a given class is an "object." Thus, to use objects, we first define a class (type) and can then declare as many objects (instantiations) of that class as we need.

The instantiation of a class as an object is achieved by calling a constructor. This is a special function of the class, which returns an object of the class type after creating the necessary storage space and performing any initialisation required. It is also possible to delete an object from memory by calling a destructor. This function is much like a constructor, but instead of creating an object, the destructor releases the object's storage space. Thus, objects are created by a constructor, and are destroyed by a destructor.

A class may contain many elements. Just as a structure can have any number of fields, so a class can have any number of "member" data fields (fields belonging to that class definition). However, unlike a structure, a class can also have any number of member functions (also referred to as "methods" or "operations"). Additionally, classes (and C++ structures) are able to declare which members are visible outside an object ("public" members), and which are hidden and visible only to other class members ("private" members). This latter feature provides the encapsulation concept mentioned above.

Additionally, it is possible for a class definition to have it's origins within another, less complex class. The new class, which "inherits" it's "form" (data fields) and "behavior" (methods) from the "base" class, is said to be "derived" from that class. This hierarchical derivation can be based on other derived classes to create a chain of ever more complex classes. Classes may also be derived from more than one base class in a single step, inheriting form and behavior from all base classes, which is termed "multiple inheritance."

Finally, if several derived classes all inherit their form and behavior from a single base class and if objects of these similar-seeming classes may all be referred to by a single name, then it is possible to use that name to denote any one of these objects. Each object so denoted is then able to respond to a similar set of operations in different ways. This concept is called "polymorphism" and the reader is referred to section 8.2 for a clarifying example.

2.3 Multitasking

The DBMES combines OO with the concept of multitasking. Multitasking allows the programmer to treat separate procedural components as independent units (or "tasks") which appear to run simultaneously. In reality the tasks all run sequentially, but portions of the execution of one task are interleaved with portions from other tasks, giving the illusion of concurrent execution. This technique requires the presence of a multitasking kernel to handle the various needs of such a system. In the Distributed Processing for Combat Systems task, all multitasking is handled by VRTX (cf. section 2.1 above).

VRTX provides many facilities for handling all aspects of multitasking. Amongst these are calls to the kernel to handle task, memory, communication, I/O and timing control aspects of the system.

Tasks can be in one of four states: executing, ready, dormant, or nonexistent. Execution is the state in which a task has control of the CPU and is actually running. Ready indicates the task is available to use the CPU and is awaiting its turn. Dormant means the task cannot execute until it receives information from some source. Nonexistent means the task is not visible from the kernel's point of view. All tasks have an ordinal number or priority which determines how great a share of the execution time the task gets. The higher a tasks priority, the more often it executes in preference to other, lower priority tasks.

Tasks need to be able to communicate with each other. VRTX provides four means to carry out this communication, which are described below.

- Queues hold multiple integer-sized messages in a first-in, first-out (FIFO) ordered, fixed-length buffer. Queues provide the primary method of transferring data between tasks. Tasks which "pend" on a queue (attempt to receive a message from the queue) do so in either FIFO order (the first to pend gets the first arriving message), or in priority order (the highest priority pending task gets the first arriving message). Although mostly used for handling messages in FIFO order, it is possible for queues to handle messages in a last-in, first-out (LIFO) order, and thus behave as a "stack".

- Semaphores are integers, holding non-negative values, which are assigned to correspond to a particular system resource (such as a memory location). When a semaphore's value is zero, access to the corresponding resource is denied to any task which consults the semaphore (the resource is "locked"). If a task does gain access to the resource (the semaphore was not zero), the task will decrement the semaphore to indicate that the resource is in use. When a task completes its operations on a resource, it increments the corresponding semaphore to indicate this (the resource is then "unlocked", or "released"). Semaphores are thus used to prevent conflicts arising from multiple tasks all accessing the same system resource simultaneously.

- Event flag groups are 32-bit sized integers. Each bit of the integer is treated as a flag, and can be either "set" (with a value of 1), or "cleared" (with a value of 0). When a particular event occurs, the corresponding flag bit is set. Event groups allow tasks to notify other tasks that a particular event has occurred, with a minimum of memory wastage.

- Mailboxes provide single integer-sized message buffers, much like single element queues, but requiring less memory. The messages handled by a mailbox must be nonzero, as a zero value in a mailbox indicates that no message has arrived at that location. Mailboxes can be used to provide the facilities of any of the above three

mechanisms, but are less 'neat' about it. They are more efficient in that there is less memory devoted to the mechanism than with the other devices.

Communication is a two-way process, in that one task must send the information, and another must receive it. The two operations need not occur simultaneously, but if a receiving task attempts to read from a buffer which has no message available, it will shift from the executing state to the dormant state until the information is placed in the buffer.

VRTX also provides facilities to handle memory management, for controlling the timing aspects of the executing and ready tasks, and for character input and output. The kernel handles the system resources which include the tasks, communication buffers, and interrupts to the operating system.

Note: It is possible to mimic the actions of all of the VRTX facilities by writing them directly into the code. It may even be cheaper with respect to memory usage. The advantage of using the VRTX facilities is that they are all protected against the potential problems which might arise should several tasks attempt to use these facilities simultaneously. This may not be the case with similar functions which are "hand-written" or taken from the standard libraries. If VRTX facilities are not used, the behavior of the system becomes unpredictable and error-prone.

2.4 Combining Object Orientation and Multitasking in DBMES

In combining the concepts of Object Orientation with Multitasking the DBMES identifies five components of the system to be separate entities. These are abstracted to become objects, and are multitasked so that they run concurrently. These multitasked objects can be referred to as "active objects," "processes," or simply "objects". The five objects are the **User**, **Sampler**, **Transaction**, **Database** and **Media**.

User is an interactive application which allows an operator to interface with the system. **Sampler** is a module which periodically consults the **Database** object to obtain performance data for evaluation purposes.

Transaction handles the collections of database operations which must be treated as a single group (eg. a series of updates may occur which must be treated as if they were a single operation, in that either all of them take place, or none do. These are also known as "atomic transactions"). **Database** is a process which maintains a local image of the global database. The local image is a component of the entire database, and it is stored on a particular node. The global database is a combination of all the local images, one for each node on the network. The **Database** process also provides the operations which allow the local image to be accessed. These two processes together implement a Distributed Database Manager (DBM) which is described in more detail in section 2.5 below.

The **Media** object handles all the aspects of communication between the processes. This communication can take place within a single node (ie. amongst the objects on that node, or "local" to the node), or across the network to objects on a "remote" node (by using the FDDI interface).

For a more technical description of combining the VRTX and Turbo C++ packages, see [4] and Appendix I of this document.

2.5 Single Record Two-Phase Commit

The DBMES was developed to be used with any DBM protocol. Ultimately it will be used to evaluate the ALDM protocol, but as this was still under development at the time of the evaluation package's construction another, simpler protocol was created. This protocol, like ALDM, provides facilities for a Two-phase Commit update, but only for a single record. Two-phase Commit is an update technique which guarantees reliability in the update, ie. all copies, through a system of "handshaking" messages (acknowledgement of requests from the receiver and vice versa), will be updated. This simpler DBM, which is not yet complete, is now referred to as the Single Record Two-Phase Commit (SR2PC) and is described below.

SR2PC provides both a performance update and a reliable update facility, as does the ALDM. Performance updates in SR2PC behave as would any other, simply broadcasting the update request and then moving on to whatever operation follows. The reliable update of SR2PC uses a different mechanism to ALDM, based on locking access to database records via a semaphore mechanism provided by VRTX (cf. section 2.3). **Semaphore** is a class built around the concept of simplifying the interface to the VRTX-provided semaphore control functions, and by making each semaphore an object, this is readily achieved. The object stores a unique identifier which is required by the VRTX functions in its own structure, and uses this in calls to the VRTX functions from the object's member functions. The object thus hides the necessity of the identifier from the programmer who is using the **Semaphore** object class.

When SR2PC updates a record reliably, it first broadcasts a "Prepare" message. All nodes with copies of the data record should then receive that message and determine if they are able to currently access that record or not. If access is permitted, the node's DBM will lock the record to prevent another (conflicting) update from taking place simultaneously, and then the DBM will reply with a "Ready" message. The locking of a record is done by a "pend" operation on a **Semaphore** object linked to that record (which is managed by the **Database** object mentioned above). The Ready reply is sent back to the node which issued the Prepare message, which counts the number of replies. If all nodes with copies respond before a timeout period expires, then an "Execute" message is broadcast to all copies which, upon receiving that message, conduct the update and then unlock the record by a "post" operation on the **Semaphore** object. If the initialising node does not receive all the replies before timeout, it broadcasts an "Abort" message to cancel the update. All nodes with copies of the record receive this message and simply unlock the **Semaphore** object. A further description of the messages and the class hierarchy of these is given in section 8.1.

Should a participating node fail to receive an Execute or Abort message due to failure of the initiating node, or loss of network communications, then the update is lost to the participant. The initiating node, on re-entering the network, will not complete the update for that node, and the system will fail because the participants will not unlock the record. This flaw will need to be addressed as part of further work on the SR2PC. If a participating node itself fails, then only that node will fail to update. The update is completed on all other nodes, and when the failed node re-enters the network it will attempt to copy the entire database from remote nodes to it's local database. At this point the update will be implicitly carried out on the returning node, by copying the updated record from another node. In this way the database consistency is maintained.

2.6 Summary of Work Completed to Date

Within the first stages of the D^{3}R project, work has concentrated on the facilities that need to be provided by the system to support dynamic reconfiguration of the network following a failure. These include a dynamic, logical view of the network, allowing the program to detect network partitions, to conduct reliable update transactions with several participating nodes and to perform system specific operations. Other work included the implementation of an object type which interfaced to VRTX's Event Group functions, providing a consistent communication facility interface.

Furthermore there needed to be some form of interface to the user to provide choices for the system configuration parameters that would affect how experiments could be run, without requiring recompilation to effect these choices. These would primarily be functions called from the User object, prompting the operator to enter a value which would then be checked before setting the appropriate parameter.

Other work involved streamlining elements of the DBMES and the GDP which could be improved. The suggested modifications arose during the familiarization stage of the early work, and were adopted later. This work provided an opportunity to become familiar with the new packages and with the DBMES' design before beginning any major work on the new protocol.

3 USER INTERFACE

The initial work has proceeded in a number of stages. The first stage involved modifying the **User** class definition and the class' constructor to allow the program user to select values for parameters such as *node_number*, *offset_time* and *update_type*. These parameters are used to configure a continuous cycle of updates, which is performed to exercise the database. The parameters are used as follows:

- *node_number* currently holds the number of the node, whose uniqueness is operator dependent. Should operator error result in two nodes sharing the same node number, the network will perform poorly. A future modification might ensure that the numbers assigned are unique.

- *offset_time* is a value from 0 to 9 which determines the delay (measured in a number of 10ths of seconds) before the first update occurs at a local node after a remote update arrives. It therefore provides a means of "staggering" the updates.

- *update_type* indicates whether the operator wishes the node to conduct either Performance or Two-phase Commit type updates. Provision for additional update types to be easily incorporated is provided.

In the second stage, the code within the **User** class constructor was modified to conduct an endless cycle of updates (with the cycle time fixed at approximately 1 second). The cycle provides a simple means of exercising the database, although it is too limited for final tests, and will be replaced by a more complex set of operations. The cycle is only broken when the operator hits the Ctrl-Z key sequence. The updates currently only occur on the database element which has the same index as the node number for this particular node, ie. Node #1 updates only Database Element (DBE) #1, Node #2 updates only DBE #2, and so forth. The reason for this is again simplicity, and the updates will eventually take place on various DBEs in the course of an application's execution.

The complexity of the actual operations of the database at this stage are of less importance than the ability to provide a constant source of operations. This will provide database activity against

which the dynamic reconfiguration will take place, and which will be unaffected by the D³R protocol.

Note: Currently the GDP displays two graphical meters which display DBE values (cf. section 4 below). These meters are keyed to respond to the first and second DBEs only, so any activity in other DBEs will not be displayed by the meters. Once again simplicity is the major factor for this decision to limit the flexibility of this facility.

4 SCREEN DEVICES

The GDP of the DBMES displays output on a graphics screen. The output is generated by objects interfacing with the Real-Time Graphics package produced by Quinn-Curtis (as mentioned in section 2.1). The output is generated by using two classes of objects described below:

- The first class is a **Text_Box**, which is simply a graphics box containing text. **Text_Box** objects are used to communicate textual data to, or provide a prompt-and-reply interface with, the user of the system. The objects of the **Text_Box** class allow the output of text strings and integer numbers into the box, and control scrolling within the box to ensure all output remains within the bounds set up for the box when first defined.

- The second class, **Update_O_Meter**, allows the definition of a graphical meter with a needle indicator and textual display of the value. These objects are used to display the values of specific database elements. **Update_O_Meter** objects can only be created, or be used to display a data value, which is reflected as a labelled point on the meter which the needle then indicates, and the textual value below the meter.

Because the Real Time Graphics (RTG) package requires specification of the number of screen elements which would be active during the life of the program, an array of predetermined size must be used to hold pointers to the viewports. DBMES has the indexes of this array hardwired into the code, and a suggested modification to the DBMES code was for the D³R program to assign these values dynamically.

To this end a new class, **Screen_Device**, was created. This became the base class from which the RTG interfaces (**Text_box** and **Update_O_Meter**) are derived. It stores a static variable *devNo* (device number) which serves as the index to the array of RTG package pointers. As each object of a derived class is created, the *devNo* is assigned to that object, and then incremented. Each object thus remembers the viewport to which it refers.

5 TEXT BOX

The Text Box is a graphics viewport generated by the RTG package in which textual output is generated. The viewport is created by invoking a class from a global object, which all processes in the program can then access. The output is written to the viewport by accessing member functions of the object, in this case using overloading of the "<<" operator.

5.1 Operator Overloading

> Operator overloading is a concept by which an operator symbol is able to mean several things. For example, "+" could refer to the addition of two integers, or two floating point numbers, or the mathematical union of two sets of numbers. "+" is then an overloaded operator. The C++ compiler provides the programmer with the capability of defining extra meanings to operators in the same way as one defines a function.

C3ISE Group/ITD has developed a new RTG-interfaced output routine which writes a text string to a graphics viewport. The function is believed to be more efficient and less prone to "noise" (ie. pixels which remained after scrolling) than the equivalent function in the original DBMES. However, attempts to incorporate this function into the program to replace the original failed and, after a week of being unable to get the program to run, the attempt was abandoned.

The failure of the incorporation occurred due to the differing paradigms used by the two programmers. While the original function was written with an object-oriented approach, the later version was not, and thus incorrect assumptions about the software environment it was intended to operate in had been made. Because of the false assumptions, the function will not work without being completely rewritten. This has yet to be done.

6 USER SUBTASKS

The next stage was to investigate the possibility of getting the **User** active object to spawn off subtasks through the use of VRTX's SC_TCREATE function, which accepts a normal function with no parameters or return type and generates a task. A task is a distinct block of program code which runs simultaneously with other tasks (in appearance), unlike normal programs which can only run through code sequentially (cf section 2.3).

Member functions of the object were found to be of the incorrect type to be made tasks unless they were declared to be static (ie. existing independent of the class within which they are declared). This meant that member data elements could not be accessed without using an explicit object name to reference them, unless they also were made static. This was done and the infinite loop of the **User** active object was removed to become the task **update_loop**. The task creation was called by the **User** object, which was prevented from terminating by self-suspension at the end of the body (ie. execution of the code is halted but it is not deleted from memory).

*Note: Because the constructor of the **User** object forms the body of a task, and VRTX deals poorly with tasks which simply end rather than become deleted (by SC_TDELETE), the body of the constructor must not be allowed to exit back to the calling function/task. Suspension is achieved by accessing the VRTX call SC_TSUSPEND, which causes the execution to halt until explicitly resumed by the SC_TRESUME call.*

7 NODE AVAILABILITY TABLE

It became clear in the process of the previous work that the system required some means of maintaining a logical image of the physical network. This view would be used by the SR2PC DBM protocol (cf section 2.5) to determine the number of copies of the database page (record) which must be updated, and by the D³R protocol to determine which nodes of the network are currently accessible. A simple table in which the node numbers of all remote nodes could be stored seemed to be the best way of implementing this.

To this end a new class, **Nodetable**, was designed. The class is dedicated to maintaining a two-dimensional array, the first element of which stores the *node_number* of a remote node, and the second, a *tag* indicating whether that node has replied to the latest inquiry for its node number.

The class holds the array as a private data element member, and its public member functions allow other sections of the program to manipulate the table. These functions are listed below:

- *Nodetable* is the constructor which initialises the node_table array to contain only the sentinel values (-1,FALSE).
- *tag_node* accepts a node number and tags the corresponding element in the array as being TRUE. It returns -1 if the node number does not appear in the table and the index of the element if it does.
- *untag_nodes* sets all tag elements of the array to FALSE in preparation for testing the network and determining which nodes still exist.
- *node_tagged* returns -1 if the node indicated by the index parameter is currently tagged, or not set to any node number. Otherwise (the element stores an untagged node which may still exist) it will return the node number stored in that element.
- *add_node* adds a node number to the array if it does not already exist in the table. If it does, the index of the node number is returned. If the number does not appear in the table, the first empty element (ie. set to -1) is replaced by the number and that index is returned. If the number is not in the table and there are no free elements, then -1 is returned to indicate failure.
- *del_node* deletes a node number from the table. If the number is present, it is reset to -1, the corresponding tag set to FALSE, and the index is returned. Otherwise -1 is returned.
- *No_nodes* counts the number of elements (in the table) which hold active node numbers, that is, node numbers which are not -1.
- *No_tags* counts the number of elements in the table which have their tag set.
- *print_tbl* prints out to the communications port/VT the table consisting of it's indexes, node numbers and it's flags.

Note: *Throughout the* **Nodetable** *class it is assumed that -1 is an indication of nonexistence or failure due to the possibility of a node having a node_number of 0. Using 0 as an indication of failure would thus be apt to cause some confusion.*

Node Availability Table	
Node Number	**Tag**
0	TRUE
2	FALSE
1	TRUE
-1	FALSE

Figure 1. Example of the Node Availability Table

The **Nodetable** class was then tested thoroughly through the use of a **User** subtask called **node_table_driver** which accepts operator input to determine which function is to be tested and then accepts operator input of the parameters for that function call. The function being tested

is then called, and results are viewed using the *print_tbl* function call. The task is no longer in use as it served only to test the new class, demonstrating that the class and object are sufficient for the purpose of maintaining the logical network image.

8 NODE NUMBER MESSAGES

Once the **Nodetable** was available, it became necessary to put it into operation. To do this it was necessary to have some form of communication between the nodes on the network which would allow the transfer of each node's number to other nodes. This was accomplished by developing two more classes which handle the Request and Reply aspects of these messages. The classes are called **NodeNo_Request** and **NodeNo_Reply**, and they are based on the message handling mechanism developed for the DBMES, which is described below.

8.1 Messages

The DBMES program is constructed of several autonomous active objects ("processes"). At times, these processes need to communicate information to each other, such as when the **Transaction** process may need to poll the **Database** object for the value of a database element which is being updated. This sort of communication is carried out by means of "messages."

Messages are objects which the processes can pass between each other. The processes do this by means of objects of the **Queue** class. The **Queue** class, in a manner similar to the **Semaphore** class (see section 2.5), serves to simplify the interface to the VRTX-provided queue communication mechanism (see section 2.3).

Queues pass pointers, which are an indirect means of referencing data in memory, including objects. The data structure resides at a particular location in the memory, and the pointer holds the memory address where the structure can be found.

To communicate a message, the processes construct a message object of the appropriate class, pass a pointer referencing that object to the **Queue** (by a POST operation) and then continue with their operations (often entering a waiting state for any reply forthcoming). The target of the message will at some point conduct a PEND operation on the same **Queue** object, and receive a copy of the pointer to the message object. This pointer can then be used by the receiving process to reference the message's operations and data.

DBMES uses seven different derived classes of message objects to carry out it's program. The classes are all derived from the base class **Message**. **Message** defines several public operations: SET_ERR, GET_ERR, GET_VALUE, and SET_VALUE, which allow access to a **Message**-derived object's data. **Message** also defines several other special operations which are PACK, PROCESS and SEND_ON. As the base class is never used to directly construct an object (ie. there is no such thing as a **Message** object), the latter three functions are defined to be dummy functions which report an error if they ever execute. They are defined to be "virtual" functions, which means that any class which is derived from **Message** can redeclare these inherited functions to behave differently.

The seven derived classes all inherit the structure of the **Message** class, including both the member data fields and operations. However, the PACK, PROCESS, and SEND_ON functions are each redefined for every new derived message class. The seven classes are **Read_Request, Read_Reply, Prepare, Ready, Execute, Abort** and **Performance**. The first two provide the facility for conducting a READ operation on the database. The next four are all used to carry out the RUPDATE (reliable update of the database), while **Performance** messages provide the PUPDATE (performance update of the database). READ, RUPDATE and PUPDATE are all described in more detail in [2].

The virtual functions all serve similar purposes in each **Message**-derived class. The various PACK functions create a data structure of the type **Fddi_union** (see section 10.1), and store all the object's state information to be sent across the FDDI network. The exact details of these PACK functions varies from class to class due to extra data types which each message needs to conduct its specialised purpose.

The PROCESS functions all provide the actual code which the recipient of the message will need to execute once the message has arrived at its intended destination. When this occurs the receiving process needs only to call the PROCESS function of the incoming message, using the pointer which the process obtained from the input **Queue**. The PROCESS function, and thus the operations prompted by the message object, will obviously be different for each differing message class.

The SEND_ON functions provide a means of telling the message-handling mechanisms of the **Media** process which queue to place the message object's pointer on. This is necessitated by the loss of such information when messages are passed across the network to a new node.

8.2 SR2PC Performance Update

What follows is a simple example of the use of message objects. It describes the operations conducted for a PUPDATE call to the database as provided by the SR2PC DBM protocol.

The **Transaction** process first constructs an object of the **Performance** class, which is then handed to the **Media** process. This process then calls the PACK function of the **Performance** object, feeding in the requisite parameter. This function returns an **Fddi_union** structure, which **Media** then transmits over the network. The destination of this transmission is the (remote) **Media** processes of every other node on the system. This is achieved by a broadcast message, ie. a transmission that all other nodes will receive.

On receiving the transmission, the message is reconstructed into an object, and passed by the FDDI interface to the (now local) **Media** process. This process calls the message object's SEND_ON function, which will direct **Media** to place the message pointer on the **Database** process' input **Queue** object. When the **Database** process receives this pointer it will use it to call the message object's PROCESS function. This will conduct the performance update on the local node. The operations described above take place on every node which is remote to the node initiating the PUPDATE call.

The other **Message**-derived classes, and the objects constructed from them, allow the SR2PC protocol to use similar mechanisms to those just detailed to conduct READ and RUPDATE database calls.

8.3 Node Number Messages

The two new message classes interact in the following manner:

- **NodeNo_Request** is broadcast onto the network and received by all nodes except the Originator of the request. When the request arrives the Receiver nodes all automatically lock rescheduling to prevent interference, create and transmit the **NodeNo_Reply** back to the Originator, and then unlock the rescheduling process.

- **NodeNo_Reply** is processed when it arrives back at the Originator. At this point it calls the ADD_NODE function (of the Nodetable object described above) with the *nodeNo* parameter and if this is successful then it calls TAG_NODE. If either function fails then the error is reported on the VT.

The operations of the message handling routines are quite complex due to the abstraction of the messages and their encapsulation of code within the message objects themselves. Once the methods involved are understood, however, this mechanism makes more sense as the code is far cleaner and better defined. It also allows message handling to be greatly simplified, and many different messages can be processed by a single unit of code, because of the polymorphic nature of the Object-Oriented classes of messages. In other words, the messages, which are all different in purpose and effect, can be treated as if they were of the same type and be handled far easier (cf section 2.2).

These new message classes are somewhat difficult to test independent of the D³R code because they require the use of almost all of the facilities provided by the DBMES. To create a test platform for these messages would require almost as much work as building the DBMES system itself. There is also a relative degree of correctness ensured by closely following the design of the DBMES message system which has proven to be correct in use. Thus these new classes were incorporated directly into the project. Final testing of the implementations was done following the next stage, which provided the necessary mechanisms to invoke the creation of the messages, and their processing when they arrive at the required destinations.

9 NODE NUMBER SUBTASKS

The messages which exchange node numbers need to originate from some task and be responded to by another independent task. It was natural that these two tasks be subtasks of the **User** object. The first, **nodeNo_mngr**, has to handle the management of this protocol, ie. the task of establishing and maintaining the **Nodetable** object. The second task, **nodeNo_resp**, simply acts as a responding server to handle any requests for a node_number by issuing a reply message.

The first subtask, **nodeNo_mngr**, operates in the following manner:

- Within an infinite loop, the tags for the table are all reset. The queue between the media and this task is cleared. A broadcast request (ie **NodeNo_Request**) is sent from this task which will prompt all other nodes on the network to respond with a point-to-point response (ie **NodeNo_Reply**). The manager task waits for these responses until a timeout occurs. Any **NodeNo_Reply** message arriving at this task is processed and, if it is from a new node (ie. one not appearing in the table), the node number is added to the table. In either case, the node is tagged as having replied.

- Once the timeout has occurred, the manager's table is checked to determine if any nodes it (the manager) believed to exist prior to the latest round of polling have failed to reply. If this is the case, then a non-broadcast message (again **NodeNo_Request**) is sent to that unresponsive (remote) node, and another reply is pended for. If the node replies within a second timeout period, the corresponding table element is tagged, otherwise the element is deleted from the table, indicating a network partition which has been recognised. It would be possible to, at this point in the program, initiate some form of network recovery protocol. This protocol will become the focus of the D³R project at some point in the future.

- The manager task is set to run every five seconds. This ensures that the processing time required by the task, which reduces the amount of available time other tasks have to access the CPU, is not too large. The manager executes its loop and then becomes dormant for five seconds before re-entering the loop.

The concept of the operation of the second subtask, **nodeNo_resp**, is far simpler than **nodeNo_mngr**:

- The task pends (indefinitely) on the queue which carries all **NodeNo_Request** messages to it, and when one arrives, it is processed (ie. a **NodeNo_Reply** message is created and sent back to the node which requested it, as described above).

*Note: The task of the **nodeNo_resp** could probably be handled by the **Database** active object with no modification at all due to the polymorphism of the messages. This is not done because the operatiion is unrelated to the database operations which are also handled by the **Database** object. The algorithms for these two tasks appear in Appendix II of this document.*

The testing of these two tasks was accomplished by simply checking to see that they worked correctly. This is easy to determine because one can tell which computers are nodes on the network by checking to see if the D^3R program is running on that computer. This can be compared to a display of the values in the logical table constructed by a node, and any lack of correspondence between the observed values indicates erroneous behavior. The **update_loop** task was shut out from the program (ie. simply not created) and the two new subtasks were observed in operation via reports to the VT. In this way the tasks were quickly verified to be working correctly.

10 NON-BROADCAST MESSAGES

It should be noted that the DBMES software only uses Broadcast messages. The manager algorithm described above uses a call to send a Non-broadcast message to a single node. The modifications to the DBMES software to achieve this were non-trivial, but have resulted in a very easy interface.

The **Media** class member function SEND was modified to accept two parameters, the first remaining a pointer to a **Message** and the second, an integer which would indicate the node number of the node for which the message was intended. This target node parameter, however, defaults to a -1 value (indicating a broadcast message) if no explicit parameter is provided.

All **Message** class PACK functions were modified to accept a node number parameter and incorporate that value into a new field of the **Fddi_union** called *target*. Thus each message on the network carries the destination within it.

10.1 Fddi_union

The **Fddi_union** is a special form of data storage which the program uses to facilitate access to the FDDI interface library. The FDDI interface routines are only able to handle messages which are streams of characters, while it is preferable for the rest of the program to deal with messages as collections of discrete numeric values (or "structures"). A union is a special structure provided by C++ which allows the messages to be treated as structures by the program code, and then be treated as character streams by the FDDI interface, without tedious conversions betwen the two forms.

Once such a message has been received (by any node on the network) the target field is examined. If the value is either -1 (broadcast) or matches the local node number value, the message is passed on through the rest of the GET_MESSAGE function which handles all incoming messages, otherwise it is simply discarded.

The benefit of this method is such that none of the DBMES code, which used the SEND function of the **Media** process (cf section 2.4), needed to be changed because the code operates on Broadcast messages and, lacking the extra parameter, continues to do so. Any calls needed for

sending destination-specific messages uses the same code, essentially, but carries an explicit target number rather than the broadcast default.

11 FLAG CLASS

The latest stage of the work was an attempt to get the **update_loop** task to utilise the *offset_time* variable entered by the operator at the beginning of the program. The offset time allows two nodes to perform updates on shared database variables in a cyclic "update/wait/update again" fashion, where the nodes avoid clashing attempts to update, by staggering the beginning of a node's cycle by the offset time. The **update_loop** must be informed of the time at which an update message arrives from a remote node, then delay for the amount of time specified by the offset time before entering the cycle of updates within that task. This information has to be imparted to the **update_loop** by the **Database** receiving the incoming message.

The initial attempts to use either a **Queue** or **Semaphore** object to provide the means of communication between the **Database** object and the **update_loop** task were unsatisfactory. The use of a queue is not possible because updates occur many times before the **update_loop** is even created. As each update occurs, it places a message on the queue to the **update_loop**, which quickly overflows and causes an error, because queues are fixed-length buffers. The use of semaphores is even less successful than queues. If the arriving updates post to the semaphore, to release the **update_loop** which pends on the same semaphore, the updates preceeding the critical time of interaction will have incremented the semaphore above zero. A non-zero semaphore value does not cause resource locking when pended to (cf. section 2.3), and so the update loop will not lock after all.

It was decided that the most appropriate device was the Event Flag Group mechanism provided by VRTX. This required the creation of a new class, called **Flag**. The class stores the *Flag_id* variable and the mask value required by many of the VRTX system calls dealing with event groups, the latter being a constant fixed at the time of the object's creation. The specifics of the class implementation closely follows the DBMES' implementation of the **Queue** and **Semaphore** classes.

The class provides member functions to handle the creation, deletion, clearing, checking, setting and consulting of the flag. Each function is self-contained and does it's own error reporting. Use of the class member functions is much simpler than the VRTX functions due to their limited capability (ie. only one flag per object) and thus the ability of the class to store the fixed value of the mask mentioned above.

Note: Use of the Event Group mechanism of VRTX should be undertaken with care because of a fault in the following functions: SC_FPOST, SC_FPEND, and SC_FCLEAR. These functions do not set the error code parameter when called, and consequently erroneous operations will possibly go unnoticed. However, provided they are used with care, there should be no problems in using the Event Groups.

12 OFFSET TIME

Once the **Flag** class was implemented, it was used to provide a means of communication between the **Database** object and the **update_loop** task. The **Database** receives a message and inquires into the status of the **Flag** object (called *update_flag*). If the flag is not set, the **Database** posts to the flag, thus setting it. The message is then processed as normal. At this stage, any message arriving at the Database sets the flag.

The **update_loop** task meanwhile is created sometime after the program commences. Once this has occurred, it clears the flag so that the next message arriving at the **Database** will cause the flag to be re-set. The **update_loop** then pends on the flag, with a timeout of one second to

ensure that at least one full cycle has passed, confirming that no updates which might affect the local node are being performed on the network. Either the arrival of a message at the **Database** or a timeout on the pend releases the **update_loop** from its waiting stage.

Whichever occurs, the task then delays for the number of tenths of seconds specified by the *offset_time* (cf section 3) variable. When the delay is released, the update cycle will commence and run until the program is terminated.

Once this was tested and working correctly, the code which posted to the *update_flag* was shifted from the **Database** object to the process functions of the **Performance** and **Execute** message classes. This means that the only messages which release the task pending on the flag are actual update-types. The **Database** object itself receives several other types of message which would be inappropriate triggers for the **update_loop**.

Note: This means that any new types of message which perform update type operations which may be added later should also post to the update_flag. The current implementation has no way of ensuring that this will happen and relies on the programmer implementing this event flag signal within the new message.

13 CONCLUSION

The Distributed Database Dynamic Reconfiguration (D^3R) protocol is intended to support a Distributed Database Management (DBM) protocol in maintaining database consistency in the event of network partitioning. The DBM could be the ALDM protocol, the SR2PC protocol described above (section 2.5), or some other which may be developed at a later date.

D^3R makes various assumptions about the form of the database, namely that it is divided into logically disparate "pages," and that the pages will all have a consistent page header, no matter what DBM schema is implemented, which will provide operational information about the contents of the page.

Much of the work described above is unrelated to the D^3R protocol itself. The work on the GDP (sections 4,5) has no relevance to database reconfiguration, and the DBMES-specific modifications (sections 3,6,12) are important only in that they involve the completion of the software which is being used to drive the D^3R protocol for test and evaluation purposes. General work (sections 10,11) is useful in enhancing the power of the D^3R system, but is not a requirement. Thus the only work which directly relates to the problem of database reconfiguration is the development of a Node Availability Table (sections 7,8,9).

The level of productivity that has been achieved in a short amount of time can be partly attributed some to the Turbo C++ packages' Integrated Development Environment, and some to the nature of the project itself. D^3R should continue to prove challenging and engaging for some time to come.

14 RECOMMENDATIONS FOR FUTURE DEVELOPMENT

A number of further steps are required to complete this project. These will include completing the DBMES GDP, the Single Record Two-Phase Commit update protocol, and then the Dynamic Reconfiguration protocol. Other components of the project will involve streamlining the operations of these projects to execute more efficiently with respect to speed, memory usage, and resource utilisation. Some of the recommended future steps that should be considered in the development of the project are listed below in a priority order:

Essential:

- Implementation of the Failure Detection and Recovery Protocols which should be able to dynamically reconfigure the network in the event of faults in the system. These protocols are the ultimate purpose of the D^{3}R project, and as such should take prime importance.

Immediately Required:

- The Node Availability Table is maintained by its own sub-protocol which is currently being redesigned for maximum efficiency. When this design is completed the protocol should be brought into line with the modifications required. At this time the Node Number subtasks should be objectified into an autonomous process, because there is no logical connection with them and the **User** interactive operator interface process. This may well mean the merging of the two distinct subtasks into a single unit with two modes of operation, if this is not already achieved by the new design. Such a modification would be far less wasteful of the relatively precious VRTX capabilities, which are somewhat limited for the larger system requirements we later face.

- Modification of the current Single Record Two-Phase Commit (SR2PC) protocol implementation to utilise the logical image of the network provided by the Node Availability Table. SR2PC relies upon knowing how many responses to an update request it should be collecting. This is currently assumed to be two, which might lead to erroneous operations on any network larger than three nodes.

- The SR2PC protocol currently performs some locking to ensure that read and write accessing of the database elements do not conflict. However, it appears that this scheme, as implemented, is not truly reliable and as such this area must be closely examined. If any flaws in the current stage of the protocol are detected, they must be corrected to provide a true reliable update. It would be difficult to evaluate a reliable update for correctness over a performance update if both were error prone.

- Modification of the **update_loop** to allow operator specification of which Database Element (DBE) to perform the updates upon. Furthermore, operator specification of the cycle time may also be desirable. This would allow the operations of the database exercising application to be more readily configured, by allowing this at run-time, rather than compile-time (which would be tedious).

- The implementation of a logical (not real-time) clock which would be synchronised between all nodes on the network. This will provide the possibility for time-stamped transaction management, where transactions can be ordered chronologically. If during database reconfiguration any transactions are lost, a check of a transaction log can allow them to be reissued in the order intended. Furthermore, modifications envisaged for the Node Availability Table may well incorporate the concept of recording the time of the last transmission by a node, and thus an early alert of possible node failure can be gained by this mechanism.

Highly Desirable:

- Modifying the **Update_O_Meter** screen devices to hold a tally of the number of circuits the needle has performed. This value should be displayed near the appropriate meter but not interfere with the labelling of the points around the display. This is required by the initial specifications written for the Graphical Demonstration Package (GDP).

- Modification of the current dynamic initialisation of the database to handle all it's elements instead of only one. Modify any replies for this purpose to non-broadcast messages. This would ensure that when the SR2PC protocol is fully completed, and conducting transactions on multiple records, that all database records are consistent before any operations take place on the local image. With full database initialisation available, the chances of any database errors would be greatly reduced.

Desirable:

- Reincorporating the new **Text_box** string output function into the program. Once again related to the GDP, this suggested modification would greatly improve the professionalism of the GDP's appearance.

- Some modification to the code might ensure that posting to the *update_flag* is enforced for current and future update protocols. Perhaps the flag should only be set when a specific DBE is updated, namely one which this node is also updating. This allows the system developer to simply link in a new update schema onto the database without having to worry about whether or not it complies to the requirements of the evaluation system.

- the **Flag** class is capable of handling up to 32 flags per object. Currently the number of flags per object is fixed at one, but a simple modification would make the others available, as well as giving the ability to pend on any or all of these flags. This would be a useful extension of the **Flag** class should it become necessary to have several events being monitored as a group by any object. Rather than creating *n* **Flag** objects to monitor *n* events, the creation of a single object which could monitor all *n* events (up to 32 per object) would be a far more efficient usage of memory, and quite likely be faster and more flexible.

15 ACKNOWLEDGEMENTS

I would like to acknowledge the participation of various members of the Distributed Processing for Combat Systems section of C3ISE group/ITD in the development of the D³R project.

Alan Allwright — Development of the FDDI communications interface library. Design and development of the ALDM reliable update protocol.

Stan Miller — The design of the DBX and ALDM. Specifications of the DBMES requirements. Initial specifications of the GDP. Development of specifications for, and assistance in the design of the D³R protocols.

William Roberts — Development of the DBX database and network communications protocols. Design and implementation of the DBMES. Design and development of the GDP. Work involved in the development of techniques for using the interface libraries for VRTX and RTG.

Alan Wood — Implementation of the new Text Box string output function. Initial work with the RTG package, and assistance with the GDP.

REFERENCES

1. Miller, S.J., "Distributed Processing Testbed System: First Interim Report for the DPTBS", WSRL Technical Memorandum, WSRL-TM-26/90, Sept. 1990.

2. Miller, S.J., "A Distributed Database Manager based on ADDAM", WSRL Technical Memorandum, WSRL-TM-58/91, October 1991.

3. Miller, S.J., "Distributed Database Management Evaluation Software", ERL Research Note, ERL-0635-RN, June 1992.

4. Roberts, W.J.J., "A Case Study in Object Oriented Design: Distributed Database Management Evaluation Software", CSI Working Paper, Draft Form only, July 1992.

APPENDIX I

USING THE VRTX INTERFACE LIBRARY WITH TURBO C++

My initial experience with Ready System's Versatile Real-Time Executive package involved the implementation of the Distributed Processing Testbed System. The source code for this was written with the Lattice C version 3.4f compiler. When I began work on the D³R project, I was essentially modifying code already written, using the Turbo C++ (TC++) compiler. This code already utilised the VRTX interface library, and much of the work of integrating the two packages had already been done [4]. Below I present some techniques and concepts which go beyond what is described there, based on my experiences with the project.

- The Linker option in the Integrated Development Environment (IDE) for Case Sensitive linking should be turned OFF due to the fact that TC++ keeps the lower-case names of the VRTX package function calls, and yet the library in which they reside has upper-case names.

- The TC++ IDE provides the facility of project files. These are used by the TC++ program to keep track of all the source files which comprise a particular program (one project per program). Project files of 75K size are definitely corrupt, and the file should be rebuilt from scratch. Project files should normally be somewhere around 10K unless the number of source files is very large. Somehow the IDE corrupts the project file and the preferable solution is to set the Auto-Save option for project files to OFF and remember to manually save the file after modifying it.

- Beware of calling constructors (defined with a void parameter list) with the function brackets explicitly encoded, eg.

```
ClassObj::ClassObj(void)
  {...}

ClassObj object();       //causes problems.
ClassObj object;              //NO problems.
```

This is because the first example of the call is declaring a function called `object` which returns a value of type ClassObj. The second, however, declares an object called `object` which is of the type `ClassObj`, which is the intended meaning.

The situation is different from defining and calling a function with a void parameter list, eg.

```
void Function(void)
  {...}

Function();
```

because in this example we are calling a function and are required to demonstrate that it has no parameters, whereas in the previous example we are creating an object and only implicitly calling the (zero-parametered) constructor function.

APPENDIX II

NODE AVAILABILITY ALGORITHMS

The following are the algorithms used in the project to maintain the Node Availability Table, as described in section 9.

Algorithm for the **nodeNo_mngr** task:

- loop forever:
  - the **Nodetable** flags are all reset to FALSE.
  - the **Queue** between the **Media** object and this task is cleared.
  - a request for node numbers (**NodeNo_Request** message) is sent to the **Media** for broadcast to the network.
  - the task then enters a loop consisting of:
    - a pend on the **Queue** object servicing this task, where a timeout of 20 ticks applys, and the following:
    - if any message arrives in that time with an error code of 0 (ie. no error) the message (a **NodeNo_Reply**) is processed. That is, the node is added to the table and tagged as having replied.
    - if the pend times out the loop is broken.
  - the manager then determines the number of nodes in the table, and the number of these which are tagged.
  - if the number of nodes does not match the number of tags another loop is entered which cycles through all elements of the table:
    - each element is checked via NODE_TAGGED to determine if it has responded to the request.
    - if it has not been tagged, a request is sent to poll that node alone in the hope that it still exists.
    - if the response arrives before timeout, (again 20 ticks,) the node is tagged as having replied.
    - if the timeout occurs the node number is deleted from the table.
  - the manager then delays it's rescheduling for 91 clock ticks, approximately five seconds.

Algorithm for the **nodeNo_resp** task:

- loop forever:
  - the responder pends indefinitely for a message on the **Queue** object servicing this task.
  - when a message arrives (only **NodeNo_Request**'s) it is processed via a polymorphic call to the message object's process member function.

DISTRIBUTION

	Copy No.
Defence Science and Technology Organisation	
Chief Defence Scientist)	
Central Office Executive)	1 shared copy
Counsellor, Defence Science, London	Doc Cont Data Sht
Counsellor, Defence Science, Washington	Doc Cont Data Sht
Scientific Adviser, Defence Central	1 copy
Electronics Research Laboratory	
Director	1 copy
Chief, Information Technology Division	1 copy
Chief, Communications Division	Doc Cont Data Sht
Chief, Electronic Warfare Division	Doc Cont Data Sht
Research Leader, Command, Control and Intelligence Systems	1 copy
Research Leader, Human Computer Interaction Laboratory	1 copy
Research Leader, Military Computing Systems	1 copy
Head, C3I Systems Engineering Group	1 copy
Mr D. O'Dea, C3I Systems Engineering Group (Author)	5 copies
Mr J. Schapel, C3I Systems Engineering Group	1 copy
Mr W. Roberts, C3I Systems Engineering Group	1 copy
Mr A. Allwright, C3I Systems Engineering Group	1 copy
Mr R. Driver, C3I Systems Engineering Group	1 copy
Mr A. Wood, C3I Systems Engineering Group	1 copy
Head, Program and Executive Support	1 copy
Mr S. Miller, Program and Executive Support	1 copy
Head, Trusted Computer Systems Group	1 copy
Head, Software Engineering Group	1 copy
Head, Computer Systems Architecture Group	1 copy
Head, Command Support Systems Group	1 copy
Head, Intelligence Systems Group	1 copy
Head, Information Acquisition and Processing Group	1 copy
Head Information Management Group	1 copy
Head, Systems Simulation and Assessment Group	1 copy
Head, Exercise Analysis Group	1 copy
Publications and Publicity Officer, ITD	1 copy
Media Services	1 copy
Department of Defence	
Director General, Communications and Information Systems	1 copy

Defence Intelligence Organisation	
DIDS	1 copy
DSCAN	1 copy
Mr P. Drewer, PRS IDS	1 copy
Army Office	
Scientific Adviser, Army	1 copy
Navy Office	
Director, Naval Combat Systems Engineering	1 copy
Navy Scientific Adviser	1 copy
Air Force Office	
Air Force Scientific Adviser	1 copy
Libraries and Information Services	
Australian Government Publishing Service	1 copy
Defence Central Library, Technical Reports Centre	1 copy
Manager, Document Exchange Centre, (for retention)	1 copy
National Technical Information Service, United States	2 copies
Defence Research Information Centre, United Kingdom	2 copies
Director Scientific Information Services, Canada	1 copy
Ministry of Defence, New Zealand	1 copy
National Library of Australia	1 copy
Defence Science and Technology Organisation Salisbury, Research Library	2 copies
Library Defence Signals Directorate, Melbourne	1 copy
British Library Document Supply Centre	1 copy
Defence Intelligence Organisation Research Service	1 copy
Spares	
Defence Science and Technology Organisation Salisbury, Research Library	6 copies
TOTAL	60 copies

Department of Defence

DOCUMENT CONTROL DATA SHEET

Page Classification	
UNCLASSIFIED	
Privacy Marking/Caveat	

1a. AR Number	1b. Establishment Number	2. Document Date	3. Task Number
AR-007-015	ERL-0663-RN	JAN 93	NAV 87/226.3

4. Title

AN INTERIM REPORT ON THE DISTRIBUTED DATABASE DYNAMIC RECONFIGURATION PROJECT

5. Security Classification

Document	Title	Abstract
U	U	U

S (Secret) C (Confi) R (Rest) U (Unclass)

* For UNCLASSIFIED docs with a secondary distribution LIMITATION, use (L)

6. No. of Pages: 26

7. No. of Refs.: 4

8. Author(s)

Damian O'Dea

9. Downgrading/Delimiting Instructions

10a. Corporate Author and Address

Electronics Research Laboratory
PO Box 1500
SALISBURY SA 5108

10b. Task Sponsor

NAVY

11. Officer/Position responsible for

Security.....

Downgrading.....

Approval for Release.....DERL.....

12. Secondary Release of this Document

APPROVED FOR PUBLIC RELEASE

Any enquiries outside stated limitations should be referred through DSTIC, Defence Information Services, Department of Defence, Anzac Park West, Canberra, ACT 2600.

13a. Deliberate Announcement

No Limitation

13b. Casual Announcement (for citation in other documents)

[√] No Limitation

[ ] Ref. by Author & Doc No only

14. DEFTEST Descriptors

Object oriented programming
Distributed database systems

15. DISCAT Subject Codes

1205, 1207

16. Abstract

This paper discusses the work which took place in the first phase of the development of the D^3R project. The concepts of Object-Oriented Programming and Multitasking are introduced and explained. Each stage of the project is detailed, and where relevant, concepts from the DBMES system are introduced and explained. The paper concludes with recommendations for future developments to both systems as they approach completion.

Page Classification
UNCLASSIFIED
Privacy Marking/Caveat

16. Abstract (CONT.)

17. Imprint

Electronics Research Laboratory
PO Box 1500
SALISBURY SA 5108

18. Document Series and Number	19. Cost Code	20. Type of Report and Period Covered
ERL-0663-RN	803885	ERL RESEARCH NOTE

21. Computer Programs Used

N/A

22. Establishment File Reference(s)

N/A

23. Additional information (if required)